To save your work, click the SAVE button at the bottom of this page. You can revisit this page, revise your answers and SAVE as often as you like.

To submit the assignment, click the SUBMIT button at the bottom of this page. YOU CAN SUBMIT ONLY ONCE. Once the assignment has been submitted, you can continue to view this page but will no longer be able to make any changes to your answers.

6.02 Fall 2014: Nagaraj,Pratheek B.

Problem Set 6

Dates & Deadlines

issued: Oct-25-2014 at 06:00
due: Nov-06-2014 at 23:45

Help is available from the staff in the 6.02 lab (38-530) during lab hours -- for the staffing schedule please see the Lab Hours page on the course website. We recommend coming to the lab if you want help debugging your code.

For other questions, please use the 6.02 online Q&A forum at Piazza.

Your answers will be graded by actual human beings (at least that's what we believe!), so don't limit your answers to machine-gradable responses. Some of the questions specifically ask for explanations; regardless, it's **always a good idea to provide a short explanation for your answer**.

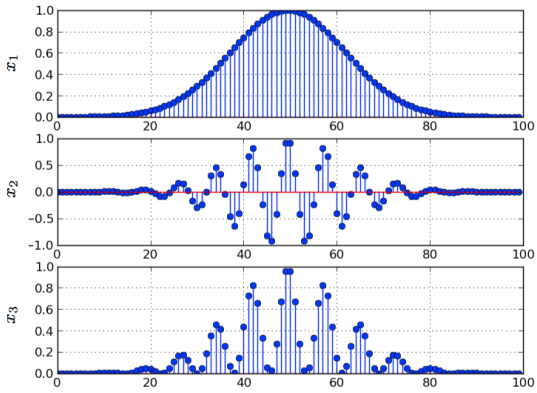
Before doing this problem set, please read the following chapters in the class notes: Chapter 13 (omit Section 13.2) and Chapter 14, which has just been rewritten in terms of the DTFT (replacing the old DTFS version). The practice problems at Modulation/Demodulation have also all been reworded in DTFT language.

Nota bene: This problem set is longer than usual, but you have two weeks to complete it. Please start early; don't wait until the day before. The programming is more involved than in the previous two PSets (though it's not onerous). The non-programming problems also require time and attention.

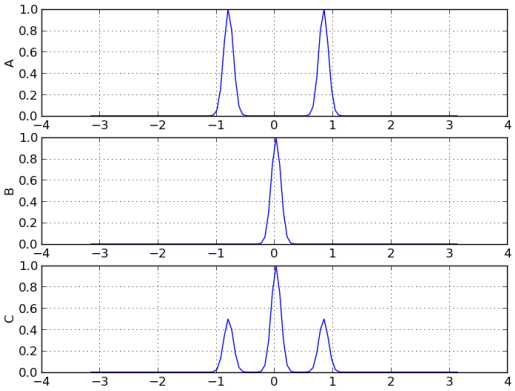
Analytical/Theory Problems

Problem 1: DTFT and Modulation (13 points)

Signals x_1, x_2, x_3 are such that $x_i[n] = 0$ for $n < 0$ and for $n \geq 100$ (so there are only 100 values that are in general nonzero, corresponding to n in the range $[0, 99]$). Furthermore, $x[49 - k] = x[50 + k]$ for $0 \leq k \leq 49$. Stem plots of the signals are shown below:



The next figure shows the plots of the magnitudes of the DT Fourier transforms of the signals (the horizontal axis extends in each case over the range $[-4, 4]$, but the DTFT is actually only plotted in $[-\pi, \pi]$). However, they are in arbitrary order, and normalized so that the maximum value in each graph is 1:



A. From the general characteristics of these signals and transforms, i.e., without any detailed computations, match the Fourier transform magnitudes to the time domain plots, explaining your reasoning in all three case (i.e., it will **not** suffice to just show the reasoning for two cases and thereby answer the third by the process of elimination!). You will be led through more detailed computations and comparisons in the following parts of this question.

```
x1 matches to B gaussian matches to gaussian
x2 matches to A sinusoidal matches to two peaks at positive and negative frequency
x3 matches to C gaussian envelope with a sinusoidal
```

(points: 3)

B. One of the DTFT magnitude plots corresponds to a signal $x_i[n]$ for which $\sum_n x_i[n] = 0$. Which DTFT is that? Explain.

The DTFT magnitude plots have all been normalized to have maximum value 1. If the plots had **not** been normalized, which of plots B and C would have had the larger magnitude at $\Omega = 0$? Explain.

Finally for this part, all the signals have the property that their alternating sum is 0, i.e., $\sum_n (-1)^n x_i[n] = 0$. What aspect of their DTFT magnitude shows this, and what aspect of the time-domain signals confirms this?

```
DTFT A corresponds to a signal that is sum x = 0

B would have a larger magnitude at omega = 0 because that is the only characteristic frequency in the
DTFT whereas for C there are other frequencies.

The DTFT in all the signals at omega=pi is equal to zero this demonstrates the alternating sum is
equal to zero. This is seen from the analysis equation when we plug in omega=pi.

We are given x[49-k]=x[50+k] for 0 <= k <= 49, which means that for every even index value we have a
equal odd index value (x[49-k]=x[50+k]) and since (-1)^n flips sign for odd and even, these quantities
perfectly cancel out across all indices and so we are left with the alternating sum equal to 0.
```

(points: 4)

C. In each case the DTFT can be written in the form $e^{-jM\Omega}C(\Omega)$, where M is some number (not necessarily integer) and $C(\Omega)$ is a sum of cosines (hence a real function of Ω). Determine M.

```
pi/4

period is 8, e^(2*pi*j*n/8) = e^(j*pi*n/4)
```

(points: 1)

D. One of the signals $x_i[n]$ is the result of modulating another one of the signals $x_r[n]$ onto the higher frequency carrier $\cos\Big(\Omega_c(n-M)\Big)$, where M is as in Part C.

Which is the modulating signal $x_r[n]$, and which is the resulting modulated signal $x_i[n]$?

Also obtain an estimate of Ω_c by inspection of the DTFT magnitude plot that goes with $x_i[n]$, explaining your reasoning. Compute the period associated with Ω_c, and compare it with the period you deduce from examination of the time domain plot for $x_i[n]$.

```
Modulating signal -> x_r = x1
Resulting Modulated Signal -> x_i = x2

omega_c ~= 0.8
omega_c = pi/4

In the modulated signal omega_c represents the shift away from the origin in the frequency domain.
Here we see that the central gaussian peak is shifted to the left and right by bout 0.8 (or actually
pi/4).

If we compare this with the period in the time domain plot we see that the signal repeats with a
period of 8 and from omega_c we have period = 2pi/omega_c = 8, which means we are on target.
```

(points: 3)

E. One of the signals $x_i[n]$ is the result of multiplying another signal $x_r[n]$ by

$$\frac{1}{2}\left(1+\cos\Big(\Omega_c(n-M)\Big)\right),$$

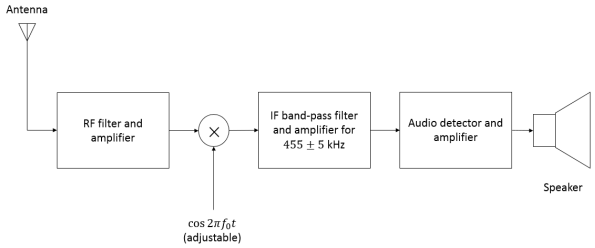
where M is as in part C. Identify x_i and x_r, and write an expression for the DTFT $X_i(\Omega)$ in terms of the DTFT $X_r(\Omega)$.

```
x_r = x1
x_i = x3

DTFT of modulating signal: X_r(omega)

DTFT of resulting modulated signal: X_i(omega) = sigma{ 1/2(1+cos(omega_c(n-M))) x[n] e^(-j omega n)}
= 1/2 X_r(omega) + 1/4 sigma{ x[n] (e^(-j omega n + j omega_c (n-M)) + e^(-j omega n - j omega_c
(n-M))} = 1/2 X_r(omega) + 1/4 e^(-j omega_c M) X_r(omega-omega_c) + 1/4 e^(j omega_c M)
X_r(omega+omega_c)

X_i(omega) = 1/2 X_r(omega) + 1/4 e^(-j omega_c M) X_r(omega-omega_c) + 1/4 e^(j omega_c M)
X_r(omega+omega_c)
```

(points: 2)

Problem 2: Tuning in a Radio Station (7 points)

Although this problem is phrased in continuous-time, it should be quite straightforward for you to do on the basis of the fundamentals you've learned in the setting of DT modulation/demodulation. If it helps you, you could think instead in terms of DT samples of the signals here, taken at some sampling rate f_s kHz, and then map each frequency f given here in kHz to a frequency of $\Omega = 2\pi f/f_s$ radians/sample. However, that's much more complication than is needed for this fairly simple and straightforward problem!



Practical radio receivers typically use several stages in their demodulation process, in order to manage various design tradeoffs. The baseband signal in AM is allowed to carry frequencies in the range of -5 kHz to 5 kHz, which is adequate for music and speech. To tune in an AM broadcast station operating at a carrier frequency of f_c kHz, we would like to select from the "radio frequency" (RF) signal picked up by the antenna only those frequencies in the ranges $f_c \pm 5$ kHz and $-f_c \pm 5$ kHz, as no other parts of the received signal are relevant. This would require building a very good bandpass filter for the antenna signal to go through, but the pass band of the filter would have to be shifted around in frequency as we tuned from one station to another—and this turns out to be tricky to do in hardware for a good bandpass filter.

Instead, what's done is to build an excellent bandpass filter whose passband is centered on a **fixed** pair of frequencies $\pm f_I$, referred to as the *intermediate frequency* or IF. For AM radio, the standard IF frequency is $f_I = 455$ kHz. This filter passes frequencies in the range $f_I \pm 5$ kHz and $-f_I \pm 5$ kHz, and effectively block out other frequencies. It can also include the amplification necessary to raise the signal level from the microvolts at the antenna to the higher levels needed to drive downstream electronics.

To move the desired radio station's signal into the passband of the IF filter/amplifier, we use the **heterodyning** idea that we've seen several times, i.e., multiply the signal at the antenna by a sinusoidal signal of appropriate frequency, f_o kHz. (In AM radio jargon, this multiplication is called mixing.) To tune in different stations, we just change the frequency of this sinusoidal signal, which is easy to do using a controllable oscillator. The common practice is to pick $f_o > f_c$; this is referred to as super-heterodyning.

A. Multiplying a carrier signal at f_c kHz with a receiver oscillator signal at f_o kHz produces a signal that is a sum of sinusoids. What are the frequencies of these sinusoids?

```
The frequencies of the sinusoids are f_o+f_c and f_o-f_c. Note that this is the temporal frequency. If
we were to use angular frequency then this would be 2*pi*(f_o+f_c)/f_s and 2*pi*(f_o-f_c)/f_s where
f_s is some temporal sample rate.
```

(points: 2)

B. What frequency f_o (in kHz, and with $f_o > f_c$) should the receiver's oscillator be set at in order to translate the signal received from an AM station operating at a carrier frequency of 580 kHz (which happens to be WTAG Worcester, in this area) into the passband of the IF filter/amplifier at 455 kHz? Explain!

```
We have f_c = 580 kHz, and since we need f_o > f_c, we select f_o-f_c = 455 kHz. Solving for f_o we
get f_o = 1035 kHz. This makes intuitive sense since we have a f_o larger than the carrier frequency
and when we mix we need to achieve the f_I frequency and so we select f_o-f_c.
```

(points: 3)

C. With f_o set as in the preceding problem, there is actually another AM station, at a carrier frequency $f_c' > f_o$, that will also find its signal translated into the passband of the IF filter/amplifier. What is f_c'?

```
f_c' > f_o = 1035 kHz. We need to locate another carrier frequency when mixed with f_o we will get the
passband of the IF filter. So, f_c' - f_o = 455 kHz, we find f_c' = 1490 kHz.
```

(points: 2)

(This f_c' is referred to as the image frequency of f_c, and the way to prevent it interfering with the desired signal centered on f_c is to do some filtering at the antenna itself, in the RF stage. Also, the FCC is careful not to assign the carrier frequencies f_c and f_c' to nearby stations.)

The output from the IF filter/amplifier can now be demodulated in the standard ways, for instance using an averaging filter if the modulating signal is always nonnegative, otherwise using a further stage of mixing in which the signal is multiplied by a sinusoid at the IF frequency. But that's a story for another day.

Python Tasks: *Sharing is Caring* - Frequency-division Multiplexing for Audiocom

In this lab—the crown jewel of the Signals part of 6.02—you will extend the Audiocom system to be able to handle multiple concurrent transmissions. In doing so, you will overcome two main challenges:

1. **Delay**: The use of a preamble to begin each packet transmission and the development of a robust method to detect a preamble in the presence of noise and channel distortions will allow us to solve one of the problems caused by an unknown delay between transmitter and receiver (Task 1). We will also need to handle the problem of an unknown phase difference caused by the unknown delay during demodulation (Tasks 3 and 4).
2. **Sharing**: To share the communication medium amongst different concurrent transmissions, one approach is to use different carriers. But how do we extract different transmissions sent at different frequencies? We need to develop a solution to this problem (Tasks 2, 3, and 4).

These challenges are tricky to overcome. They are hard enough on nice, well-behaved channels, but harder still if we were to try to get our solutions working over any real-world communication medium off the bat. A general approach to combat real-world complexities is to first design under a simplified simulation of real-world conditions. We will do that using the `AbstractChannel`, which we introduced in earlier problem sets.

By modeling the communication channel in terms of three basic abstractions—noise, delay, and the unit sample response (using our assumed LTI model)—we can try out our ideas and test them under controlled conditions. After they work here, we will run them over real-world audio channels and see whether they need further tuning or refinement, and evaluate performance. Such a method is a good principle to adopt in general when confronted with the task of getting a real system working under complex conditions.

Download the PS6 code here. As always, you should be sure to calibrate Audiocom before starting, and run any Audiocom commands in this lab with the parameters your environment requires (`-q`, `-p`, etc.).

Task 1: Preamble Detection (5 points)

Note: If you get stuck on this task, you can move on to the later tasks below using the `preamble.pyc` module we have provided (copy either `preamble_26.pyc` or `preamble_27.pyc` to `preamble.pyc`; you may also need to rename `preamble.py` to make sure Python doesn't try to compile it). You can come back to using your own `preamble.py` implementation later.

Problem: In any communication system, the transmitter may start transmitting at any point in time. How does the receiver know when a legitimate transmitter has begun? Even when the transmitter is not sending anything, a receiver listening on the medium will acquire samples. We need a way to determine that a legitimate transmission has started.

Over many communication media, this task is performed using a *preamble*. A preamble is a known sequence of bits—which are of course converted to samples—with some nice properties.

The transmitter begins its communication with the preamble. The receiver then listens for the preamble to know that transmission has begun; in other words, the preamble helps *synchronize* the transmitter and receiver.

To detect the preamble, the receiver searches its received samples for a sequence that most closely resembles the preamble sample sequence. Because of noise on the channel, the receiver will not receive the preamble sequence exactly; hence the "most closely".

Look at the file `preamble.py`. It contains the `Preamble` class. The three class variables are:

- `self.preamble_data`: The bit sequence we use for the preamble.
- `self.silence`: The number of "silent" bits appended to the beginning of the preamble. This ensures that the receiver is ready and does not miss the preamble. (The number of quiet bits doesn't matter to the receiver; it just focuses on searching for the preamble data)
- `self.preamble`: The actual preamble (silence + data)

Your task is to write the body of two functions: `detect` and `correlate`.

Implementing `detect`

`detect(`*`demodulated_samples`*`, `*`receiver`*`, `*`offset_hint`*`)`

This is the primary function for detecting the preamble in a series of demodulated samples.

This function takes three arguments: *`demodulated_samples`*, a numpy array of demodulated samples; *`receiver`*, which allows us to access some helpful parameters (more on that in a bit); and *`offset_hint`*, which is a hint provided from the caller telling `detect` to start looking from *`demodulated_samples[offset_hint]`* for the preamble sequence (`offset_hint` is a likely offset where some non-zero energy corresponding to the first "1" might occur).

`detect` should return the offset into *`demodulated_samples`* that corresponds to the start of the preamble sequence.

You will see that we have already set up some helpful local variables for you in `detect`. Your job is to complete the rest of the function.

There are many ways to design and implement `detect`, but the one we recommend is described below. Though it involves five steps, **each step should only be one or two lines of code at most**; if you find yourself writing more than that, please come to office hours because you are probably doing something wrong (or at least, inefficient) :).

1. Convert the preamble bits—stored in self.preamble_data—to a sequence of samples. The function `receiver.mapper.bits2samples` will be helpful.
2. Produce the modulated waveform, by multiplying the array of preamble samples with a local carrier at the carrier frequency. The function `sendrecv.local_carrier` will be helpful.
3. Demodulate the result of the above step. The function `receiver.demodulate` will be helpful.
4. At this point, the demodulated samples from the above step represent a sort of "ideal" sample sequence; they're the samples we'd receive in the absence of any noise on the channel. The last step is to use `correlate` to correlate these demodulated samples with the received demodulated samples. `correlate` will return the index of the sample that is most likely to be the start of the preamble.
5. Finally, you should return the index into `demodulated_samples` where the preamble is most likely to start.

Note that the array `demodulated_samples` may be too long, and searching for the preamble samples through the entire array may be too slow. To speed-up this search, we don't need to search through the entire reception because we know that the preamble is somewhere near the beginning of the reception, once the intial "silent" samples (which may have been corrupted by noise) have been dealt with. To help here, use the `offset_hint` variable. Our recommendation is that you start your preamble search from `demodulated_samples[offset_hint]` and try to find the preamble by correlating over a number of samples equal to two or three times the numper of samples in the preamble's data sequence. If you do this, be careful to return a value of `offset_hint + index`, where `index` is the result returned by `correlate`.

Implementing `correlate`

correlate(x, *y*)

This function takes two arguments: `x` and `y`, both numpy arrays. It should return the index in array `y` corresponding to the best occurrence of sequence `x`. (In effect, we're searching `y` for the sequence `x`, or at least the sequence most similar to it.)

You may assume that `len(x) <= len(y)` and return 0 if either `len(x) > len(y)` or `len(x) == 0`.

The correlation between two *equal-length* arrays, a and b, is the normalized dot product between a and b, i.e.,

$$\frac{a \cdot b}{\|a\|\|b\|}$$

You can use the functions `numpy.dot` and `numpy.linalg.norm` to help compute this value.

`correlate` should find the *maximum* normalized dot product of `x` with all subsequences of `y` whose length is equal to the length of `x`. It then returns the index (in `y`) of the corresponding subsequence.

To help you with this function, we've included a small test case in `preamble.py`. Just running

```
$ python preamble.py
```

will test your code on the arrays `[0.2, 0.4, 0.6, 0.8]` and `[0.1, -0.05, 0.05, 0.08, 0.14, 0.22, 0.4, 0.8, 0.1, 1.0]`. Your code should return the index 2, because the subsequence `[0.05, 0.08, 0.15, 0.22]` has the highest correlation among all possible choices for `x = [0.2, 0.4, 0.6, 0.8]`.

Testing

In addition to the correlation test we provide in `preamble.py`, you can test your preamble detection with the `AbstractChannel` (use the `-a` option with `sendrecv.py`). This will ensure that your preamble detection is not too far off base; if something is terrbily wrong, you won't be able to transmit data on the `AbstractChannel` (because the preamble will not be detected). However, just because your preamble detection works with the `AbstractChannel` does *not* mean it's working perfectly.

If your `preamble.py` does not work, please upload it anyway. We will try to give partial credit wherever we can.

View uploaded file

Upload your preamble.py Browse... No file selected.

(points: 5)

Interlude: The Problem with Audiocom (1 point)

Recall the modulation and demodulation steps from PS4 and PS5. To modulate, we multiply the signal with a cosine carrier of a specified carrier frequency. We did this step to match the transmitted signal to the characteristics of the communication channel, which in our case (the audio channel) is able to carry sinusoids across some frequency range. With envelope demodulation, we take the absolute value of the received samples and run the result through a simple averaging filter. The final step—which you explicitly explored in PS5—is to take the average of the middle M samples of each bit, where M is one-half the value of the samples per bit.

Suppose we use this method for **two** concurrent transmissions. Will it work? To answer this question, run the following:

```
$ python sendrecv.py -f testfiles/A -f testfiles/B -c 1000 -G 800
```

(the `-G` option specifies the frequency gap, in Hz, between the carrier frequencies of the concurrent transmissions, starting from the base specified in the `-c` option. The `-f` option tells audiocom what text to transmit (the text is stored in the passed file))

With these parameters, you should be able to transfer data, i.e., the preamble can be detected. But, did this method work? Did you receive the data that you expected to receive? Explain what happened.

(points: 1)

Task 2: Demodulation Part I - Handling Multiple Senders (2 points)

Note: If you get stuck on this task, you can move on to the later tasks below using the `demodulate.pyc` module we have provided (copy either `demodulate_26.pyc` or `demodulate_27.pyc` to `demodulate.pyc`; you may also need to rename `demodulate.py` to make sure Python doesn't try to compile it). You can come back to using your own `demodulate.py` implementation later.

Problem: We need a way to extract the messages modulated on different carriers at the receiver.

A clever way to extract multiple messages is with **heterodyne demodulation**: multiply the received samples by a local version of the same carrier sinusoid that was used by the transmitter. You can read about this in-depth in Sections 14.2 and 14.3 of the course notes.

In `demodulate.py`, fill in the incomplete line of `heterodyne_demodulator` setting the carrier to be a cosine with the specified carrier frequency. You can do this by calling `sendrecv.local_carrier` as explained in Task 1. Note that `demodulate.py` has changed, and that filtering is broken out as a separate step, which our code calls for you (i.e., you don't need to call the averaging filter function from `heterodyne_demodulator`).

Now try the following (notice we're using the `AbstractChannel` here, and that we're just trying to get *one* file transmitted successfully):

```
$ python sendrecv.py -f testfiles/B -a -v 0.1 -d het
```

You should see the contents of testfiles/B: `E pluribus unum`. Now try running the heterodyne demodulator over the audio channel:

```
$ python sendrecv.py -f testfiles/B -d het
```

It probably won't work reliably. (If it does, it was good luck; we still need to understand why you got lucky!)

To understand what's going on, let's go back to our `AbstractChannel`. Run the following four experiments, which add a delay lag (in number of samples) between transmitter and receiver, where `X` is the ratio of the sampling rate (default: 48000 per second) to the carrier frequency (default: 1000 Hz) (you'll need to fill in a numerical value for `X`):

```
python sendrecv.py -f testfiles/B -a -v 0.1 -l 0 -d het
python sendrecv.py -f testfiles/B -a -v 0.1 -l X -d het
python sendrecv.py -f testfiles/B -a -v 0.1 -l X/2 -d het
```

```
python sendrecv.py -f testfiles/B -a -v 0.1 -l X/4 -d het
```

What can you conclude from these experiments about the heterodyne demodulator and why it does not always work (but sometimes does)?

(points: 2)

Task 3: Demodulation Part II - Handling Lag (4 points)

Note: If you get stuck on this task, you can move on to the later tasks below using the `demodulate.pyc` module we have provided (copy either `demodulate_26.pyc` or `demodulate_27.pyc` to `demodulate.pyc`; you may also need to rename `demodulate.py` to make sure Python doesn't try to compile it). You can come back to using your own `demodulate.py` implementation later.

Problem: Heterodyne demodulation only works with very specific lag values. We need a demodulation method that is more general.

By now, you've probably understood that we need **quadrature demodulation** (see Chapter 14), and not a purely heterodyned demodulation, to handle an arbitrary delay lag between transmitter and receiver. Taking advantage of Python's natural support for complex numbers, we will multiply the received waveform by a complex exponential waveform at the carrier frequency.

In `demodulate.py`, fill in the incomplete line of of `quadrature_demodulator` by setting the carrier to be the quadrature version of the local carrier.

Now run the following tests with the `AbstractChannel` (as before, `X` = sampling rate / carrier frequency):

```
$ python sendrecv.py -f testfiles/B -a -v 0.1 -l X -d quad
$ python sendrecv.py -f testfiles/B -a -v 0.1 -l X/2 -d quad
$ python sendrecv.py -f testfiles/B -a -v 0.1 -l X/4 -d quad
```

And now try the `AudioChannel`:

```
$ python sendrecv.py -f testfiles/B -d quad
```

Does your quadrature demodulator (with the above commands) correctly receive data? Explain why it works or doesn't (as always, by "works" we mean reliably sends data).

(points: 1)

Now add in a second transmission:

```
$ python sendrecv.py -f testfiles/A -f testfiles/B -a -v 0.1 -l X/4 -G 800 -d quad
```

Does this quadrature demodulator correctly receive data? What if you added a third transmission, and a fourth (just use testfiles/C, testfiles/D to add more files)? Explain your answer. Your explanation should be informed by playing close attention to the **signal spectrum** arriving at the receiver and **after** passing through the quadrature demodulator and the averaging filter. As long as you are using quadrature demodulation, you can produce these spectra by adding the `-g` option to `sendrecv.py`.

(points: 2)

If your `demodulate.py` does not work, please upload it anyway. We will try to give partial credit wherever we can.

Upload your demodulate.py: Browse... No file selected.

(points: 1)

Task 4: Filtering (4 points)

Problem: We need a filter that is more effective at correctly extracting only the part of the received signal intended for it, and ignoring everything else.

One filter that may work is the **sinc low-pass filter** (see Section 12.2.2, example 6, as well as Section 13.1). An ideal sinc has an infinite number of non-zero components in its unit-sample response, which is hard to implement on a computer in finite time. So we will build a sinc filter of some pre-defined length, say `L=50`.

Your job is to implement the following function in `filter.py`:

`low_pass_filter(`*`samples`*`, `*`channel_gap`*`, `*`sample_rate`*`)`

This function takes an array of samples, `samples`, which we would like to apply the low-pass filter to. The parameters `channel_gap` and `sample_rate` each specify the corresponding

parameters (which you can set for the system with the `-G` and `-r` flags as you've seen). This function should return the samples after filtering.

Inside `low_pass_filter`, your first task should be to pick a good cutoff frequency for the filter. You can approach this task with the following steps:

1. How should you pick the cutoff frequency of the sinc low-pass filter? You need to set this to a value that will support multiple concurrent transmissions. Factors that you may want to consider include:
   - The sample rate
   - The frequency gap between successive carriers

   You don't necessarily need to incorporate all these factors in your cutoff frequency setting.
2. Generate the unit-sample response, h, for the sinc filter of length `L=50`. Do that by creating a numpy array, `h`, that corresponds to the sinc filter of cutoff frequency `omega_cut`, for $-50 \leq i \leq 50$, including at $i = 0$. Though i goes from -50 to 50, your `h` should be indexed starting at zero (i.e., `h[0]` corresponds to $i = -50$, `h[1]` corresponds to $i = -49$, etc.).
3. Then, convolve `samples` with `h`, and return that result. You may implement the convolution yourself or use `numpy.convolve`; the returned result should be a numpy array.

Further notes that may be helpful:

- You can use `numpy.sin` and `numpy.pi` or `math.pi` to access sine values and π, respectively.
- You will likely need to convert continuous-time frequencies into a discrete-time frequencies. That was covered in Lecture 13.

While testing, you should continue to use the `-g` option, which will allow you to see the frequency spectrum of the demodulated samples after filtering (you will get a separate graph for each channel; annoyingly, you need to close the first graph before the second will appear, so you might want to save the graphs).

Run your quadrature demodulator on multiple transmissions and see if you can demodulate properly. You should be able to transmit all four files successfully:

```
$ python sendrecv.py -f testfiles/A -f testfiles/B -f testfiles/C -f testfiles/D -G 600 -d quad -t lp -a -v
0.2
```

And for the grand finale, run the following:

```
$ python sendrecv.py -f testfiles/A -f testfiles/B -d quad -t lp -G 600
```

(You may need to change the frequency gap to a different value in the -G option above, and may also need to increase samples per bit to a value larger than the default of 256.)

In a quiet environment, you'll find that you can transmit as many six files at once, both with the `AbstractChannel` and the `AudioChannel`. (If you try six, brace yourself for the level of sound that will be emitted! And please do not try six in the lab; everyone around you will hate it.) If you're able to get this many files or more, send us an email and we will personally congratulate you. :)

If your `filter.py` does not work, please upload it anyway. We will try to give partial credit wherever we can.

Upload your filter.py: Browse... No file selected.

(points: 4)

Interview points for PS6:

(points: 4)

You can save your work at any time by clicking the Save button below. You can revisit this page, revise your answers and SAVE as often as you like.

Save

To submit the assignment, click on the Submit button below. YOU CAN SUBMIT ONLY ONCE after which you will not be able to make any further changes to your answers. Once an assignment is submitted, solutions will be visible after the due date and the graders will have access to your answers. When the grading is complete, points and grader comments will be shown on this page.

Submit